PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 17/30 — A1

(11) International Publication Number: WO 00/45304

(43) International Publication Date: 3 August 2000 (03.08.00)

(21) International Application Number: PCT/US00/02139

(22) International Filing Date: 28 January 2000 (28.01.00)

(30) Priority Data:
60/117,909 29 January 1999 (29.01.99) US

(71) Applicant: OBJECT DESIGN, INC. [US/US]; 25 Mall Road, Burlington, MA 01803 (US).

(72) Inventors: BACON, Stephanos; Object Design, Inc., 25 Mall Road, Burlington, MA 01803 (US). FEIN, Peter; Object Design, Inc., 25 Mall Road, Burlington, MA 01803 (US). RABIN, Paul; Object Design, Inc., 25 Mall Road, Burlington, MA 01803 (US). RESNICK, Michael, L.; Object Design, Inc., 25 Mall Road, Burlington, MA 01803 (US). WEINREB, Daniel; Object Design, Inc., 25 Mall Road, Burlington, MA 01803 (US).

(74) Agents: COHEN, Jerry et al.; Perkins, Smith & Cohen, 30th Floor, One Beacon St., Boston, MA 02108–3106 (US).

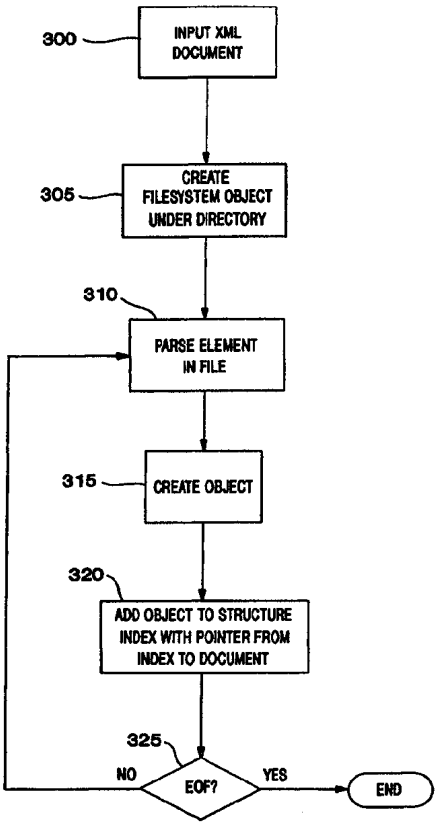
(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: DATABASE MANAGEMENT SYSTEM WITH CAPABILITY OF FINE-GRAINED INDEXING AND QUERYING

(57) Abstract

A transactional store of XML documents maintains an index that can be queried at a fine grain level. The documents may be any type of XML documents (300), which encompasses a broad range of data types. A document object interface that enables an index to be built by breaking down a received document into its component parts. The index is a decomposed form of each document with pointers into the documents. The XML grammar enables the invention to maintain relationships inside documents and across documents so that the querying can be done more quickly than if documents had to be parsed with each query (310). The index also allows documents to be queried in a consistent maner. An update language integrates a new document into the existing index (320), as well as removes documents to be deleted and updates the database when a change is made.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

DATABASE MANAGEMENT SYSTEM WITH CAPABILITY OF FINE-GRAINED INDEXING AND QUERYING

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority of U.S. provisional applications Serial No. 60/117,909 entitled, "XML Database Management System" filed January 29, 1999.

FIELD OF THE INVENTION

This invention relates generally to databases and more particularly to database management systems using XML as a primary data model.

BACKGROUND OF THE INVENTION

With the improvements in computer capabilities, there has been an exponential increase in data stored in databases. Once stored, data needs to be accessible, preferably quickly and to a fine degree of granularity. The vast amounts of data stored in databases drives a need for data retrieval to be efficient and specific. The finer the degree of granularity, the more specific a data retrieval can be, and the more specific the data retrieval, the less time and other resources that are wasted by the data requestor.

eXtensible Markup Language (XML) was developed by the SGML Editorial Board formed under the auspices of the World Wide Web Consortium (W3C) in 1996. XML is a pared-down version of Standard Generalized Markup Language (SGML), designed especially for Web documents, however XML is also useful for object-oriented databases. XML enables Web designers and other programmers to create customized tags to provide functionality not available with Hypertext Markup Language (HTML). For example, XML supports links from a point in a document to multiple other documents, as opposed to HTML links, which can reference just one destination each.

XML is based on the concept of documents composed of a series of entities or objects. Each entity can contain one or more logical elements. Each of these elements can have

certain attributes (properties) that describe the way in which the element is to be processed. XML provides a formal syntax for describing the relationships between the entities, elements and attributes that make up an XML document, which can be used to tell the computer how it can recognize the component parts of each document. In other words, XML describes a class of data objects called XML documents and also partially describes the behavior of computer programs which process the documents. By construction, XML documents are conforming SGML documents.

XML differs from other markup languages in that it does not simply indicate where a change of appearance occurs, or where a new element starts. XML sets out to clearly identify the boundaries of every part of a document, whether it be a new chapter, a piece of boilerplate text, or a reference to another publication.

The entities, or objects, of XML contain either parsed or unparsed data. Parsed data is made up of characters, some of which form the character data in the document, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. The markup portions of an XML document are also called "tags." XML provides a mechanism to impose constraints on the storage layout and logical structure. A software module called an XML processor is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the application.

XML is designed to make it easy to interchange structured documents over the Internet. XML files always clearly mark where the start and end of each of the logical parts (called elements) of an interchanged document occurs. XML restricts the use of SGML constructs to ensure that fall back options are available when access to certain components of the document is not currently possible over the Internet. It also defines how Internet Uniform Resource Locators can be used to identify component parts of XML data streams.

By defining the role of each element of text in a formal model, known as a Document Type Definition (DTD), users of XML can check that each component of the document occurs in a valid place within the interchanged data stream. The DTD declares each of the permitted entities, elements and attributes and the relationships between them. An XML DTD allows computers to check, for example, that users do not accidentally enter a third-level heading without first having entered a second-level heading, something that cannot be checked using the HyperText Markup Language (HTML) previously used to code documents that form part of the World Wide Web (WWW) of documents accessible through the Internet.

Unlike SGML, however, XML does not require the presence of a DTD. If no DTD is available, either because all or part of it is not accessible over the Internet or because the user failed to create it, an XML system can assign a default definition for undeclared components of the markup. XML enables users to bring multiple files together to form compound documents where illustrations may be incorporated into text files, and the format used to encode each illustration provides processing control information to supporting programs, such as document validators and browsers.

XML is formal language that can be used to pass information about the component parts of a stored document. XML is flexible enough to be able to describe any logical text structure, whether it be a form, memo, letter, report, book, encyclopedia, dictionary or database. XML is a common syntax for expressing structure in data. Structured data refers to data that is tagged for its content, meaning, or use. For example, whereas the <H1> tag in HTML specifies text to be presented in a certain typeface and weight, an XML tag would explicitly identify the kind of information: <BYLINE> tags might identify the author of a document, <PRICE> tags could contain an item's cost in an inventory list, all the way down to <DOGFOODBRAND> if that is the level of detail required.

XML separates structure and content from presentation, allowing the same XML source document to be written once, then displayed in a variety of ways, e.g. on a computer monitor,

within a cellular-phone display, or a personal digital assistant. Therefore, an XML document can outlive and move beyond the particular authoring and display technologies available when it was written including the Internet.

The highly detailed structure and flexibility of XML makes it ideal for database applications. XML requires detailed marking of a stored document in a way that provides a rich, yet simple, data model for storing data in a way that makes the data easy to access even at a fine-grained level.

There are a number of types of databases but currently large amounts of data are commonly stored in relational databases (RDBs). A relational database management system (RDBMS) stores data in the form of related tables. Relational databases are powerful because they require few assumptions about how data is related or how it will be extracted from the database. As a result, the same database can be viewed in many different ways. An important feature of relational systems is that a single database can be spread across several tables. This differs from flat-file databases, in which each database is self-contained in a single table.

In relational databases, a join operation matches records in two tables. The two tables must be joined by at least one common field. That is, the join field is a member of both tables.

Object oriented data bases (OODBs) directly map computational objects to persistent data structures. The fundamental argument in favor of OODBs, as compared to relational data bases (RDBs), relies on a measure of data complexity. The need for OODB representations quickly increases with data complexity, which depends on a number of factors including: 1.the absence of a permanent and unique identification of the represented objects; 2.the existence of many-to-many relationships; 3.the evolution of data structures; 4.complex relationships between object instances and object features; 5.the existence of a large variety of data types; and 6.complex hierarchical structures between data types.

RDBs were originally designed to represent "simple" data types, such as customer names (strings) or product prices (numbers) and simple relationships that can be represented in tables (matrices). On the other hand, OODBs were designed to represent complex and "natural" data structures that often exist in the "real" world. Such objects tend to have a complex hierarchical structure with distributed and sparse relationships between objects and object features. As the complexity of object structures increases, OODBs have tended to scale up "linearly" with object relationships, whereas RDBs tend to require an exponentially increasing amount of join tables.

RDBs may be used to store complex objects, however there are problems. When a database representation does not match the computational representation, there is an impedance mismatch. Such a mismatch may generate substantial problems in RDB representations of complex objects including: excessively large amounts of memory necessary to represent complex data relationships into join tables; significant performance problems during queries; and exponentially increasing difficulty in maintaining and extending RDB tables as data complexity increases.

Relational and hybrid object-relational database management systems have been developed, however they are not well suited for component-based computing applications. RDBMSs and ORDBMSs deconstruct objects into tables for storage, and in the process, lose the explicit relationships between objects. RDBMSs and ORDBMs have to reconstruct the relationships each time data is retrieved by executing CPU-intensive joins. Database performance degrades exponentially as the number of tables involved in a join increases.

RDBMSs were designed long before distributed architectures emerged. Database processing is server centric so queries, joins, stored procedures, and triggers occur on the server and as a result, database requests from distributed components across the network must all queue up at a central database server, which results in server bottlenecks and excessive network traffic.

RDBMSs do not integrate well with object-oriented languages. A mismatch exists between RDBMSs which manage simple data as rows and columns, and components written, for example in Java or C++, which manage data as objects. It is the developer's responsibility to address this mismatch by writing mapping code to translate between objects and rows. Mapping code adds no value to solving problems and often composes 25 - 40% of an application, increasing time-to-market, decreasing flexibility, and creating an on-going maintenance burden.

Relational joins limit performance and scalability. For example, components written in Java and C++ are rich entities of interrelated object data. In order for an RDBMS to manage component data models, it must perform time-consuming joins to relate the data at run time. This severe performance penalty prevents large-scale deployment or requires exorbitant hardware investments.

RDMBSs do not support rich, user-defined, data types. RDBMSs support alphanumeric data types only, and they cannot be extended to support new user-defined data types. Object-relational hybrids allow you to define new data types, which means you have to wait for joins, and extensibility is still limited by the need to write code to convert component object modes to tables.

It remains desirable to have a database management system with a data model that is richer than the relational model, i.e. without having to do join computations, while allowing flexibility in the model so that the schema can be changed easily.

It is an object of the present invention to provide a method and apparatus to manage data stored in existing databases.

SUMMARY OF THE INVENTION

The problems of storing and accessing data to a fine degree of granularity in a database having a flexible data

model are solved by the present invention of an XML database system.

A transactional store of XML documents maintains an index that can be queried at a fine grain level. The documents may be any type of XML documents, which encompasses an broad range of data types. The invention makes available a document object interface that enables an index to be built by breaking down a received document into its component parts. The index is a decomposed form of each document with pointers into the documents. The XML grammar enables the invention to maintain relationships inside documents and across documents so that querying can be done more quickly than if documents had to be parsed with each query. The index also allows documents to be queried in a consistent manner. An update language integrates a new document into the existing index, as well as removes documents to be deleted and updates the database when a change is made. The index with its pointers into the documents at a fine-grained level enables the documents to be manipulated as well as queried at the sub-document level.

The present invention together with the above and other advantages may best be understood from the following detailed description of the embodiments of the invention illustrated in the drawings, wherein:

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of a computer system using a data management system according to principles of the invention;

Figure 2 is block diagram of the computer system of Figure 1 showing more components of the data management system according to principles of the invention;

Figure 3 is a block diagram of a document in the file system;

Figure 4 is a first structure index according to principles of the present invention;

Figure 5 is an example document in the filesystem;

Figure 6 is a flow chart of constructing a structure index from a document;

Figure 7 is a structure index of the document of Figure 5; and

Figure 8 is a flow chart of parsing and executing an update request according to principles of the invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Figure 1 is a computer system 10 having a data management system 15 according to the principles of the present invention. The computer system 10 has a client server architecture. A client 20 and the database management system 15 are connected by a network of some type, e.g. a LAN, a WAN, or the Internet. In the present embodiment of the invention the client 20 is a thin client, i.e. it is a small embeddable component that runs in the same process as a client application.

The database management system 15, also referred to as the server, has the functional modules that process data. The database management system 15 has an admin (administrative) server 30, XML stores 35, 40, a configuration database 45, and caches 50, 55. The administrative server 30 and the caches each have a client interface 60. In the present embodiment of the invention, the client interface 60 is a COM interface (Component Object Model, a model developed by Microsoft Corporation). In alternative embodiments, other interface standards may be used.

The admin server 30 can perform all the functions of the database management system 15 including operating a default cache. The admin server 30 can launch caches as required by configuration data and client requests. The admin server 30 will be described more fully below.

The XML stores 35, 40 have any data of interest to the users of the system. The XML stores 35, 40 may store any type of XML documents. Those items that are not XML documents are stored as blobs.

The caches 50, 55 are processes having associated caches of data from the XML stores 35, 40.

The configuration database 45 generally contains the configuration information. That is, it maintains information about the XML stores 35, 40 and the caches the admin server 30 is supposed to launch for particular processes. The configuration database 45 also contains security information.

In operation, a user program on the client 20 sends a request to the admin server 30 for data. The admin server 30 examines the configuration database 45 and sends the configuration information which includes the caches which are involved in the operations, e.g. storing and querying, and whether the caches have already been created. Using the configuration information, the client 20 then connects to existing caches. The admin server 30 creates any new caches which are needed and then the client 20 connects to those. The client 20 then routes requests for data to the appropriate cache for the particular operation.

The server 15 provides persistence of the data objects stored in the XML stores. Persistence is defined as the preservation of a data object and its relationships through transactions involving the data object. The database management system manages data as XML objects.

Figure 2 shows the functional components of a server process. The admin server 30 and the caches 50, 55 of Figure 1 are all server processes and the server process of Figure 2 illustrates any one of them. The server process has an engine 100, an engine API 110, and a JAVA API 120, as well as the COM API 60 described above with regard to Figure 1. The engine 100 is the core of the database management system 15 and has a number of processes including an XML parser, a query engine, an update engine, a DOM implementation, a file system, a configuration component, and a security component. The XML parser parses XML. The query engine implements XQL. The update engine implements an update language which has an XML grammar and creates a document having attributes, etc. The update engine interprets the update language and uses the information contained therein to update other documents. The

DOM is an implementation of the Document Object Model, the interface that defines the mechanisms for accessing data in the XML database. The file system is a logical view of the XML store and acts as the API for manipulating files. The configuration component is used for creating and deleting XML stores and caches in the configuration database. The security component is related to the file system and the configuration component. The security component controls whether a particular user can perform a particular operation on the system.

The engine is a dynamic link library wrapped in APIs through which the outside applications link to the applications in the engine. The engine API 110 exposes the applications inside the engine to the applications outside the engine. In the present embodiment of the invention, the engine API 110 is implemented in C++. The Java API 120 is a translation of the engine API 110 into Java. The client interface, or COM API, allows the engine to be accessed remotely, e.g. by a remote client. The COM API gives access to the server via a remote procedure call (as implemented in COM/DCOM (DCOM is Distributed Component Object Model, an extension of COM and also developed by Microsoft Corporation)). The client interface permits no DOM level access to the engine. The Java API is used by user-written components called server extensions that run in the context of some server process, i.e. in either the admin server 30, or the caches 50, 55. Server extensions are named as elements in an XML store file system and can be invoked via the COM client API.

Figure 3 is a diagram of a typical document in a filesystem of the present invention. The document 200 is hooked to the filesystem directory 205 via a filesystem object 210 that has the document name and permissions. The document 200 is arranged in a hierarchical "tree." In the present invention, the component parts are for example tags, elements and text substrings according to the XML standard. The XML standard encourages rigorous detailing of document elements.

This in turn enables the decomposing of an XML document into its component parts in a consistent manner.

Figure 4 shows first example of a structure index that is built from a document of the type shown in Figure 3 according to principles of the present invention. The structure index is the result of parsing and decomposing the document. This example shows how the element "foo" is indexed. The document having the element is parsed and "foo" is decomposed into its component parts which are then indexed into a structure index that is organized as a hierarchical tree, different from the hierarchical tree in the filesystem as shown in Figure 3.

Figures 5-7 illustrate the decomposition of a document in greater detail. Figure 5 shows a directory having a filesystem object having a document in XML format. XML enables structured data to be put into the format of a text file. The document in Figure 5 is laid out in rigorous XML format where each element of the document is described by a tag (and the appropriate end tag).

Figure 6 is a flow chart of the process of parsing a document to create a structure index. A document in XML is taken as input, block 300. A filesystem object is created inside the directory of stored files, block 305. A first element in the file is parsed, block 310. An object is created, block 315, and then added to the structure index with pointers from the structure index tree to the document, block 320. The process then checks if the end of the file (EOF) has been reached, block 325, and stops if it find the EOF. If not, the process parses a next element in the file and continues.

Figure 7 shows the index for the directory, filesystem object and document shown in Figure 5. The document is a child of the filesystem object which in turn is a child of the directory. The document is indexed according to its component parts. The tag a is at the top of document. B is a child of a. C and d are children of b. "q" is a child of c and takes the value "5." The document is broken down in the index to each of its component parts. One can arrive at the value of q efficiently using the index path.

Additionally, because the document is broken down into its component parts, and each part is indexed, it is simple to do a search on all c's, or all a's in the directory rather than having to search through every document to find a "c" or an "a."

In order to access the data, a data path is used. The data path is associated with content indices. The content indices are the value texts and word/text indices. That is, the data is accessible using a path, and an index can be added to the path. For example, one can use the following to find a customer in Boston: /customers/customer[address/city = "Boston"] where /customers/customer is the path and address/city is the key.

Update Language

The update language creates new elements in the database, removes elements from the database and updates the database. The update language follows the following principles. The update language is a declarative, or "rules-based" language using an XML grammar. The update language provides all functions available via procedural document object model (DOM) that defines what attributes are associated with each object and how the objects and attributes can be manipulated. The functions of the update language include creating new nodes in the document tree, new trees, new attributes and entire documents. The update language operates on single nodes and bulk nodes. The content and attributes of existing nodes can be modified. Also, nodes and attributes can be removed from the document tree. The update language operates across documents in the database. The update language can store the grammar as a parse tree and optimize the grammar based on queries performed. The implementation of the update language represents an update message internally as a parse tree, analyzes the internal representation, re-formulates it into an equivalent form for more efficient processing thereby optimizing it. Updates to the database can be sent as web-based requests. The update language makes updates re-entrant, that is the update language allows concurrent updates on unrelated data in the database. The update language can

target pieces of elements (e.g. words of text) for update and removal.

The basic update element is "xlnupdate" which identifies a request as being a database management system update. This element has a namespace reference as well as a required version number, indicating the version of the update grammar supplied. The following is an example of a database update:

```
<xml version="1.0">
<xlnupdate version="1.0"
xmlns:xlnup="http://www.odi.com/excelon/update">

  content of update request

</xlnupdate>
```

The content of the update element "xlnupdate" is one or more of these elements (also called "rules"):

- <foreach>
- <create>
- <remove>
- <update>.

"xlnupdate" is allowed only the above-listed top level elements as direct children. An update request can be processed in the context of an existing document. If an update request has document context, this is known at the time that processing begins. If an update request does not have such context, the specific document, or documents to be acted upon are provided in the update request itself. Any document context provided inside the update overrides the current context. The top-level elements in an update request are processed in order. If no further information is provided, the scope of the update is the current document context.

The <foreach> element is a basic looping construct with a mandatory select pattern. The rules contained in the element

apply to all matching patterns. There may be more than one element inside, and elements are applied sequentially.
For example,

```
  <foreach select="pattern" >
    <from/>
    <update/>
or
    <create/>
or
    <remove/>
    <foreach/>
  </foreach>
```

For the nested <foreach> elements, the query is performed once, and sub-queries are scoped to a lesser set. Also, nesting semantics are well defined for dealing with newly-created, or modified sub-elements.

The <create> element creates a new entity in the system. <Create> supports elements, attributes, and comments and other XML objects such as processing instructions (PIs), as well as files and XMLStores. The <select> attribute is optional, but only if working in the context of a <foreach>; otherwise it is mandatory. If <select> is used while inside a <foreach>, it is evaluated relative to the current target element. The target of the <select> attribute must be an element (can be root). It can select into existing text, only in the case where a <text> element is being created.
For example,

```
 <create [select = "pattern"]>
 <from/>
<element location={"firstchild" | "lastchild" | "before" |
"after"
       | "insert" } [ movefrom="pattern" | copyfrom="pattern"
] >
```

Element content goes here, and is created, uninterpreted. No content if <movefrom> or <copyfrom> attribute exists.

```
    </element>
    <attribute name="attrname">attribute value</attribute>
    <comment location={"firstchild" | "lastchild" | "before" |
"after" |
"insert" } >
     text body of comment;
    </comment>
    <text location={"before" | "after"}> text to be inserted.
```

If the target is an existing element, the text is simply inserted as content as a first child node. If there is existing text, the existing text is concatenated onto the new text.

If the target is inside existing text, the location attribute is used to determine if the new text goes before or after the target.

For example,

```
    </text>
    <pi name="name of PI">value of PI</pi>
    <document filepath="path_to_file"
            [type="{directory|file}"]
            [storename="XMLStore_name"]
            [overwrite="{true|false}"]
            [exclusive={"true|false"}] />
    <!-- other types of nodes can be added, as desired -->
  </create>
```

A single <create> element contains only one of the following child elements: <element>, <attribute>, <document>, or <comment>.

Attributes of the <create> element further include "where", "movefrom," and "copyfrom." If either "*from" attribute is set, it specifies a node, or possibly a set of nodes, that are to be used as the source of the creation. If the attribute is "movefrom" a MOVE operation is performed on the target nodes. If the attribute is "copyfrom," the source

is simply copied. If either of these attributes is set, then the content of the "element" element must be empty.

The "where" attribute tells the statement where, relative to the create target, to put the new element. The values of "where" are: "insert" which creates a new child of target element, that child inserted anywhere; "firstchild" which creates a new child of target element, the new child to be inserted first; "lastchild" which creates a new child of target element, that child inserted last; "before" which creates a new sibling, and inserts it before a target; "after" which creates a new sibling, and inserts it after a target.

The <document> attribute in the <create> element creates a new document, of type directory of file. The current XMLStore and file context is used for resolving the path for the new document, unless a storename attribute is used. The type defaults to "file."

The <remove> element removes an entity from the system. The <remove> element acts on elements, attributes, comments, and XML objects (e.g. PIs), as well as files and XMLStores. The "select" attribute under the <remove> element is optional and has the same semantics as in <create>, above, except that it can target an attribute.

For example,

```
<remove [select = "pattern"] >
  <from/>
</remove>
```

The code in the example above simply removes the target document, element or attribute specified.

The <update> element modifies an entity, in place. This element may replace attribute values, or content, including subtrees of elements. The select attribute under <document> is optional and has same semantics as in create, above, except that it can target an attribute. An optional "newname" attribute allows an update to change the name of the targeted object, effectively modeling a <replace> attribute including name.

For example,

```
<update [select = "pattern"]>
  <from/>
  <element [newname = "newname for element"]>
    New content for element goes here.
  </element>
  <attribute [newname="newname"]>newvalue</attribute>
  <text>new text to replace target</text>
  <comment>new text for comment</comment>
  <pi [newname="newname"]>new pi content</pi>
  <!-- add other objects as required -->
</update>
```

The following are elements used inside the top-level elements. Contextual elements help give context to the update, and identify nodes for update. Creation elements tell the update what type of XML structure to create or modify. Contextual elements are "document" and "modify." Creation elements are <element>, <attribute>, <text>, <comment>, and <pi>. The use of <element>, <attribute>, <text>, <comment>, and <pi> are consistent between <update> and <create>.

The <document> element is always optional, and helps scope a given update operation with respect to XMLStore and filename. It provides support required to deal with multiple documents and stores in a single request. For example,

```
<document filepath="path_to_file"
          [type="{directory|file}"]
          [storename="XMLStore_name"]
          [overwrite="{true|false}"]
          [exclusive={"true|false"}] />
```

<element>, <attribute>, <text>, <comment>, and <pi> are simple elements used only by the <create> and <update> elements, telling <create> and <update> what type of data to create.

The "type," "overwrite," and "exclusive" attributes are only used for the <create> element. The "type" attribute

tells the update language processor to create a directory or a file. The "overwrite" attribute tells the update language processor what to do if it sees a file or directory or other object with the same name. The "exclusive" attribute tells the update language processor whether or not the access to the file is read or write.

The "from" element is used to control selection of target elements and documents for update. Its syntax is:

```
<from select="pattern">
  [<document/>]
</from>
```

More than one <from> element may exist inside a single, top-level element. <from> must be a direct child of a top-level element, such as <update>. When more than one <from> exists, the parent element operation is performed over the set of all <from> elements. Essentially, <from> is a way to scope a single operation (e.g. <create> or <update>) across multiple documents and even XMLStores, without making the top-level element semantics overly complex. All of the top-level elements have an optional "select" attribute. If this attribute is used, the element must not contain any <from> elements. The use of the "select" attribute at this level, and <from> elements is mutually exclusive.

Implementation

Figure 8 is a flow chart of the parsing and execution of an update request according to principles of the invention. The update in XML format, block 400, is parsed into a document object model (DOM) tree, block 405. The DOM tree is traversed, creating an execution list of objects for each major operation (e.g. foreach, update, etc), block 410. Each executable operation is modeled as an object of the same type. The end result is an ordered list of linked execution objects. Each execution object has its own specific state, based on the arguments passed in. A first major element is executed, block 415. If the major element has nexted elements, block 420, the nest objects are executed, block 425. Elements which allow

nesting (foreach, update) have their own execution lists, which are traversed as part of their own execution. The major elements are executed sequentially until the end of the execution list is reached, block 430.

The following are some examples where

-> represents an execution list link, and

| V represents the head of an execution list.

A simple example is:

```
list
  |
  V
  create_object -> update_object -> remove_object
```

The following example presents a more complex list, including a <foreach>:

```
list
  |
  V
  create_object -> foreach_object -> remove_object
                        |
                        V
                        update_object -> create_object
```

The following example presents an even more complex list, with nesting inside the update object:

```
list
  |
  V
  create_object -> foreach_object -> remove_object
                        |
                        V
                        update_object -> create_object
                            |
                            V
                            foreach_object -> create_object
                                |
```

V

(foreach object's list)

The <foreach> and <update> objects each has its own execution list, which is traversed once for each target element. In the case of nested updates, there is only one object; however, if one of the nested element is a foreach, then it will loop.

For parameters, it would be useful to cache pre-parsed templates, perhaps by creating a new file system type, like an update stored procedure. General abstraction principles should be followed. Specifically, the implementation of the execution tree should be as independent of the update grammar as possible. This provides for maximal code reuse if/when other updates are supported, as well as good engineering.

It is to be understood that the above-described embodiments are simply illustrative of the principles of the invention. Various and other modifications and changes may be made by those skilled in the art which will embody the principles of the invention and fall within the spirit and scope thereof.

What is claimed is:

1. A method for updating a computer database, comprising the steps of:

providing an update request, said update request having a plurality of elements, each element in said update request delimited by a tag;

parsing said update request;

building a document object model in response to said parsing step;

traversing said document object model to form an execution list of objects; and

executing said execution list sequentially thereby updating the database.

2. The method of claim 1 wherein said providing step further comprises providing said update request over a web-based computer network.

3. The method of claim 1 wherein said update request is in XML format.

4. The method of claim 3 wherein said parsing step further comprises interpreting XML tags.

5. The method of claim 1 wherein building a document object model further comprises defining attributes and associating said attributes with objects.

6. The method of claim 5 wherein said building step further comprises defining rules by which said attributes and said objects are handled.

7. The method of claim 1 wherein said traversing step further comprises forming an execution list of a plurality of major execution objects wherein at least one of said major execution objects has at least one nested execution object.

8. The method of claim 7 wherein said executing step further comprises executing said at least one nested execution object in sequence with said at least one major execution object to which it belongs.

9. An apparatus for updating a computer database, comprising:

an update request, said update request having a plurality of elements, each element in said update request delimited by a tag;

a parser for parsing said update request;

a document object model builder for building a document object model in response to said parsed update request;

means for traversing said document object model, said means for traversing to form an execution list of objects from said document object model; and

an execution list processor for executing said execution list of objects,

whereby the database is updated as said execution list of objects is executed.

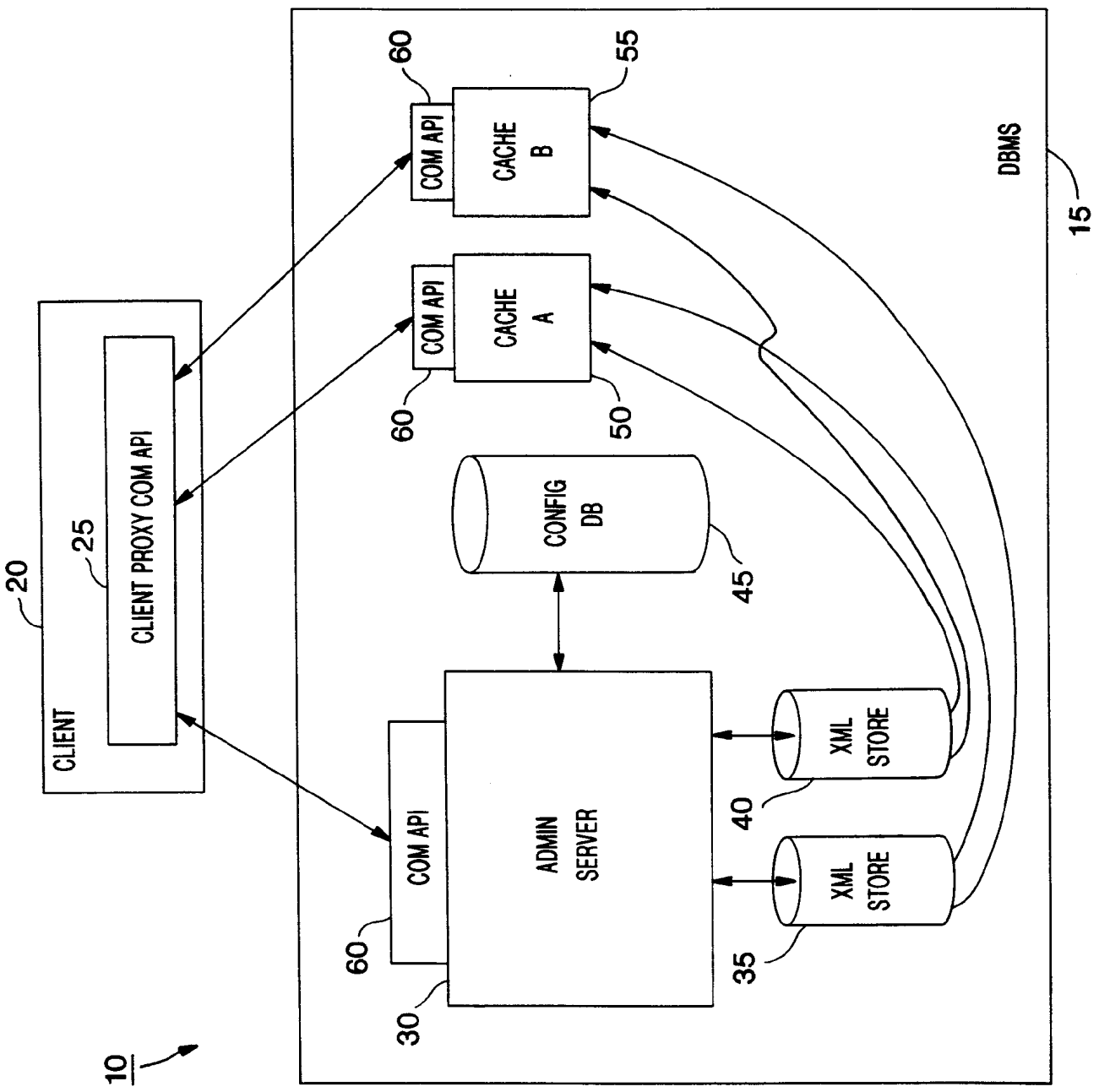

10
CLIENT
20
CLIENT PROXY COM API
25
DBMS
15
COM API
60
CACHE B
55
COM API
60
CACHE A
50
CONFIG DB
45
COM API
60
ADMIN SERVER
30
XML STORE
40
XML STORE
35

FIG. 1

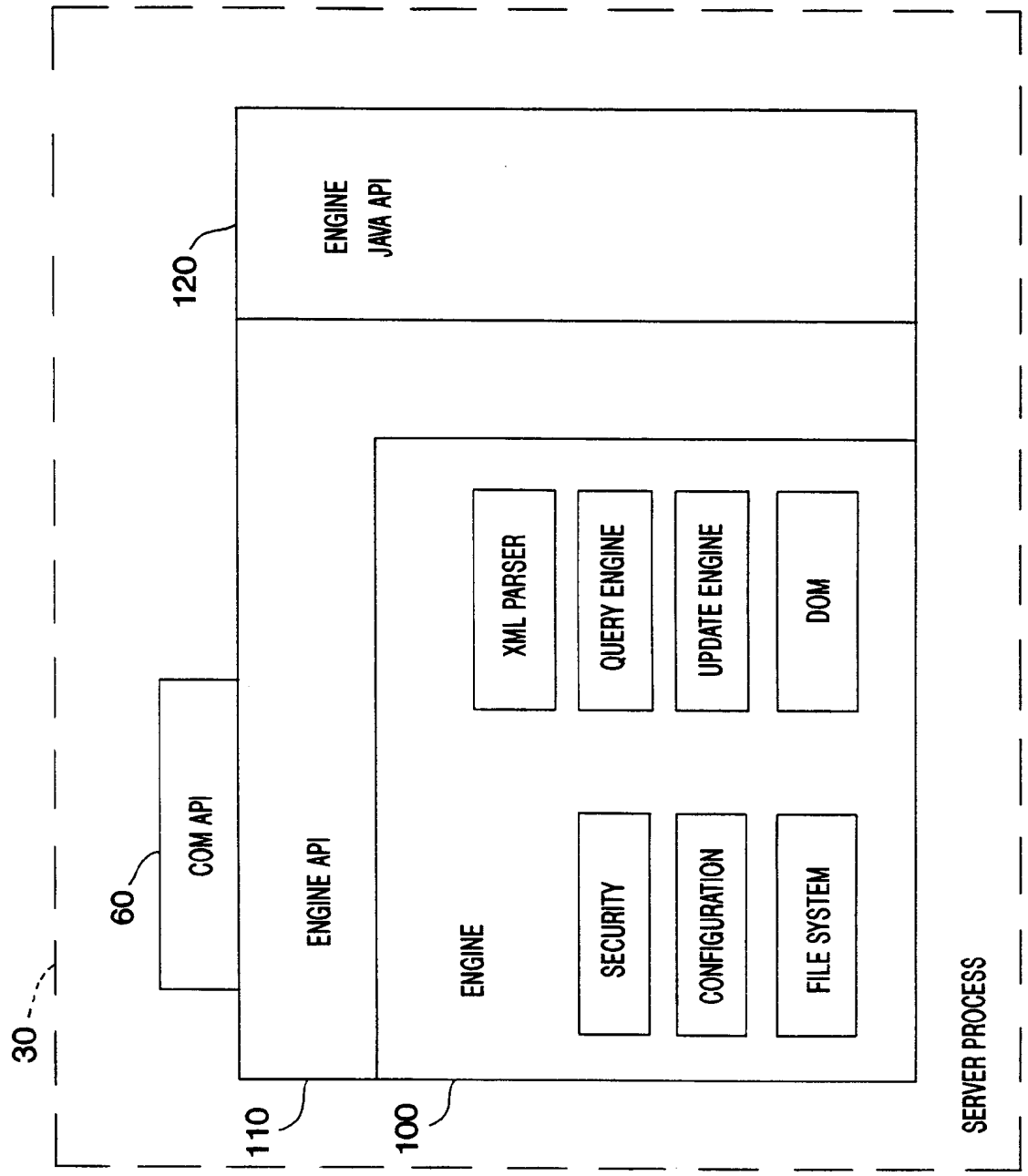

30
60
COM API
ENGINE API
110
ENGINE
100
XML PARSER
QUERY ENGINE
UPDATE ENGINE
DOM
SECURITY
CONFIGURATION
FILE SYSTEM
120
ENGINE
JAVA API
SERVER PROCESS

FIG. 2

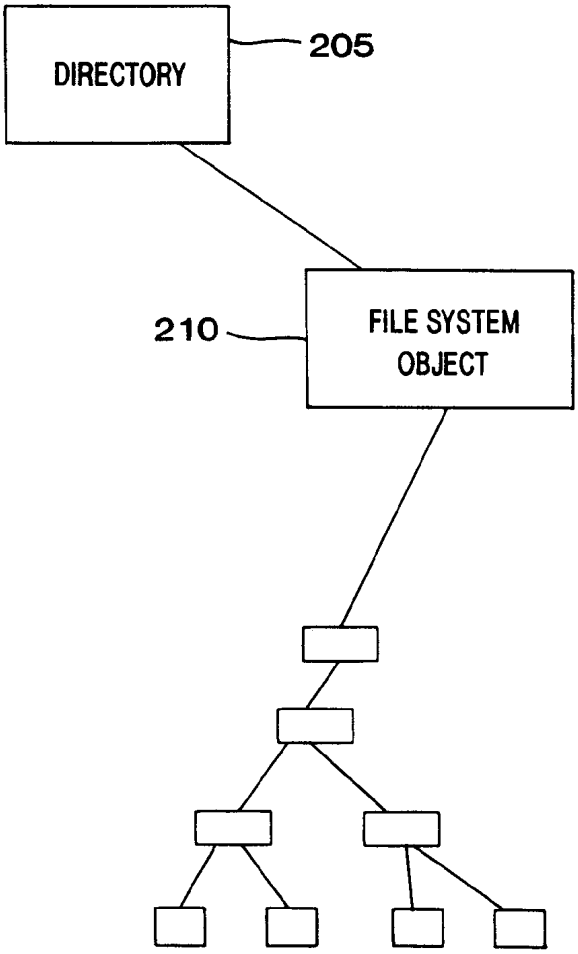

DIRECTORY
205
210
FILE SYSTEM OBJECT

FIG. 3

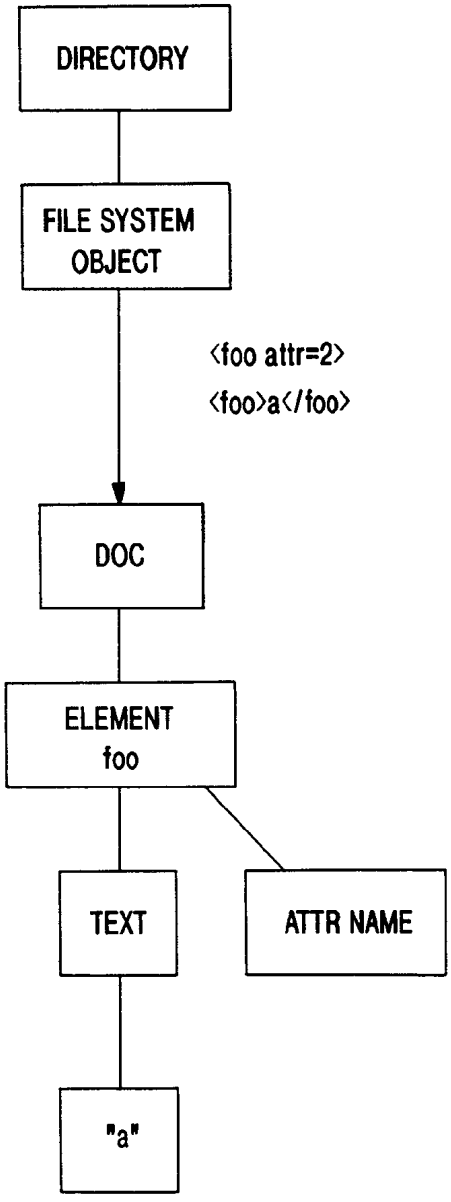
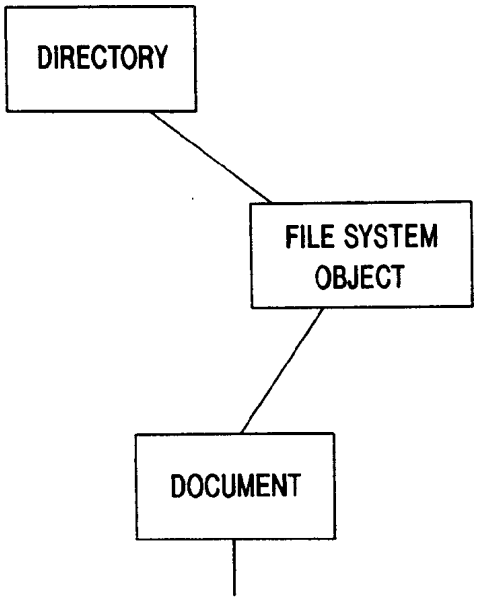

DIRECTORY
FILE SYSTEM OBJECT
<foo attr=2>
<foo>a</foo>
DOC
ELEMENT foo
TEXT
ATTR NAME
"a"

FIG. 4




```
<a>
  <b>
    <c>
      <q> 5 </q>
    </c>
  </b>
  <b>
    <d> </d>
  </b>
</a>
```

FIG. 5

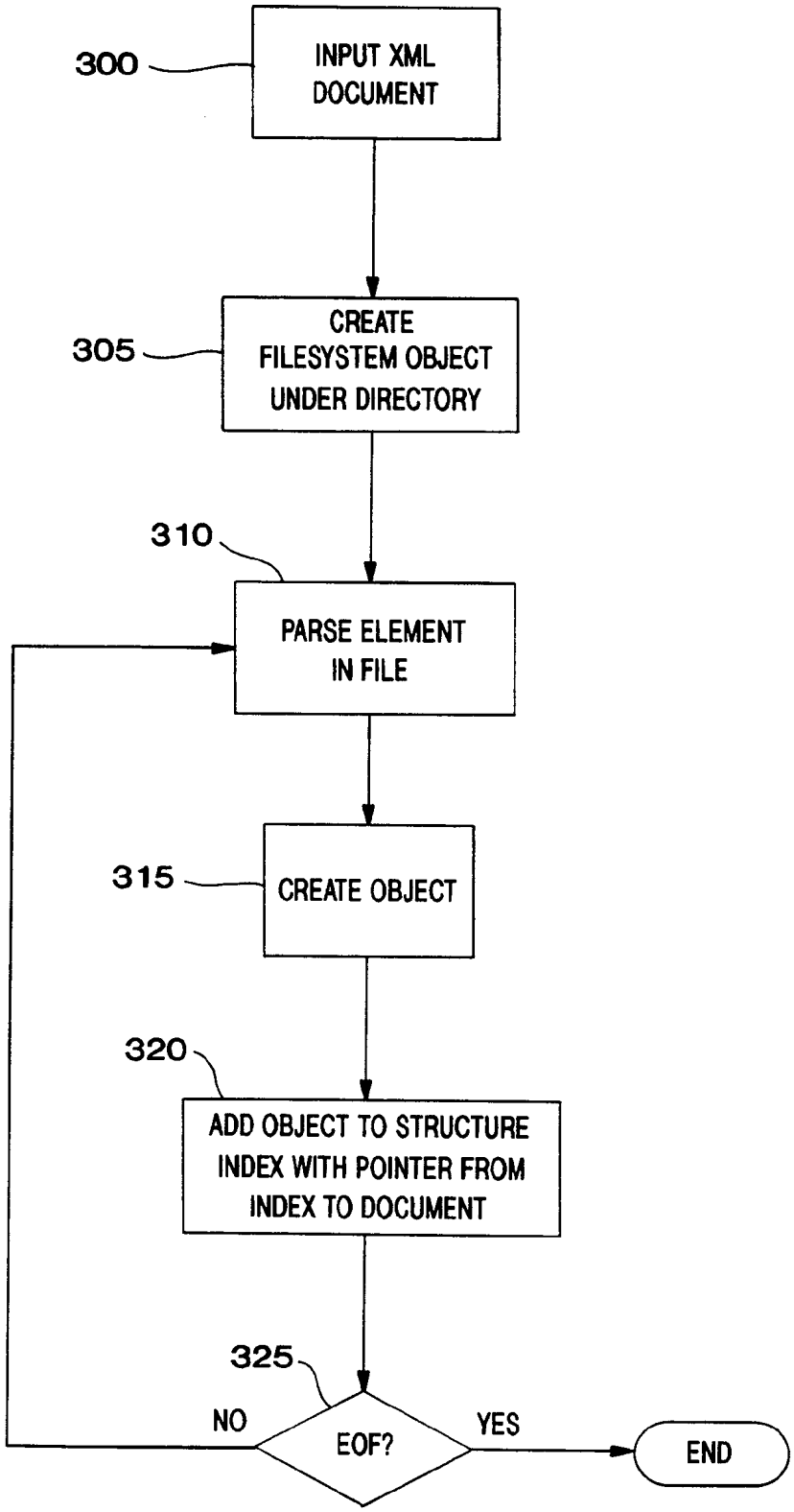

300
INPUT XML DOCUMENT
305
CREATE FILESYSTEM OBJECT UNDER DIRECTORY
310
PARSE ELEMENT IN FILE
315
CREATE OBJECT
320
ADD OBJECT TO STRUCTURE INDEX WITH POINTER FROM INDEX TO DOCUMENT
325
EOF?
NO
YES
END

FIG. 6

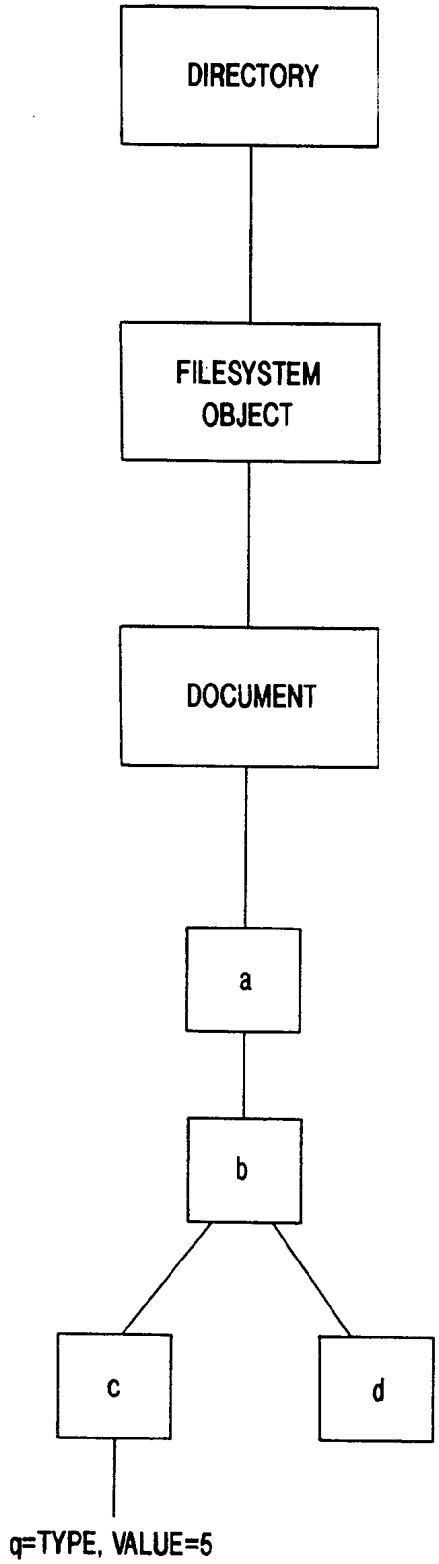


FIG. 7

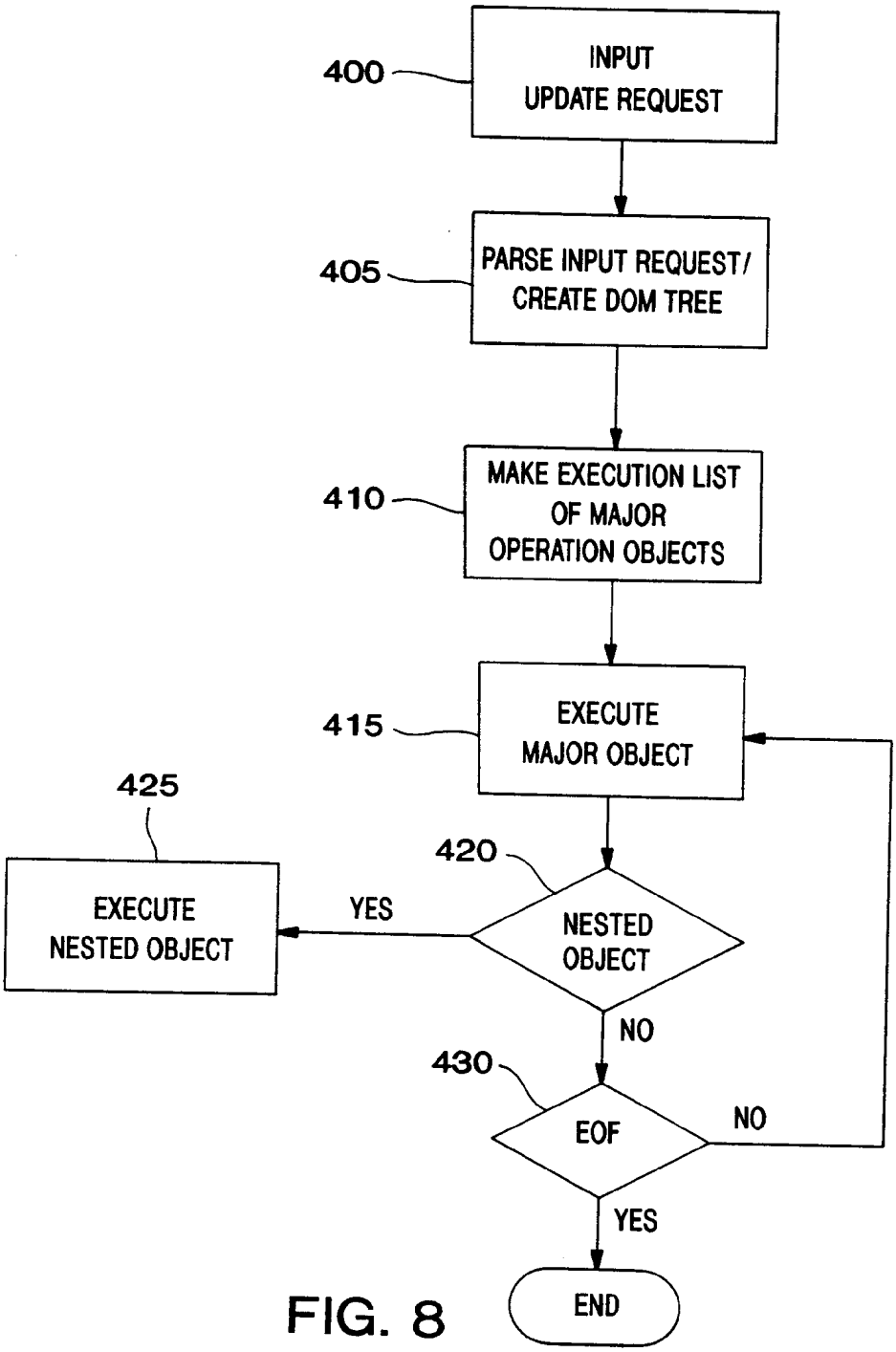

400
INPUT
UPDATE REQUEST
405
PARSE INPUT REQUEST/
CREATE DOM TREE
410
MAKE EXECUTION LIST
OF MAJOR
OPERATION OBJECTS
415
EXECUTE
MAJOR OBJECT
425
420
EXECUTE
NESTED OBJECT
YES
NESTED
OBJECT
NO
430
EOF
NO
YES
END

FIG. 8

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/02139

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) :G06F 17/30
US CL :707/3, 10, 103, 104

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 707/3, 10, 103, 104

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EAST

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y,P	US 5,970,490 A (MORGENSTERN) 19 October 1999, column 9, line 4 - column 24, line 15.	1-9

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 10 APRIL 2000	Date of mailing of the international search report 16 MAY 2000
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer KIM YEN VU Telephone No. (703) 305-4393

Form PCT/ISA/210 (second sheet) (July 1998)